# ChronOS Linux: A Best-Effort Real-Time Multiprocessor Linux Kernel

Matthew Dellinger, Piyush Garyali, and Binoy Ravindran
ECE Dept., Virgina Tech
Blacksburg, VA 24061, USA
{mdelling,piyushg,binoy}@vt.edu

## ABSTRACT

We present ChronOS Linux, a best-effort real-time Linux kernel for chip multiprocessors (CMPs). ChronOS addresses the intersection of three problem spaces: a) OS-support for obtaining best-effort timing assurances, b) real-time Linux kernel augmented with the `PREEMPT_RT` patch, and c) OS support for CMP-aware real-time scheduling. While each of these spaces have been studied in the past, their intersection, which has strong problem motivations, was previously empty. Best-effort timeliness targets real-time applications with run-time uncertainties and resource overloads, and optimizes collective application timeliness — as specified by the application. ChronOS directly supports the implementation of best-effort real-time schedulers on CMPs, in addition to others, in the global and partitioned scheduling disciplines. ChronOS extends the **PREEMPT_RT** Linux patch, and thus provides full kernel preemptibility and retains stock Linux features. We validate our claims by reporting on the implementation of a suite of best-effort and non-best-effort CMP schedulers on a quad-core AMD Phenom platform.

## Categories and Subject Descriptors

E.2.1 [**Embedded Software and Tools**]: Real-time operating systems and middleware

## General Terms

Design

## Keywords

Real-time, task scheduling, Linux

## 1. INTRODUCTION

Chip manufacturers are increasingly using chip- and system-level parallelism to improve performance by manufacturing a new generation of processors with multiple cores on a chip, called chip multiprocessors (or CMPs). Consequently, the design of multiprocessor real-time scheduling algorithms has become important in order to allow real-time applications to take advantage of CMPs. This also motivates the need for operating systems, which must support CMP scheduling.

The real-time problem space includes application contexts where key aspects of application behavior—e.g., task arrivals, execution times, resource access patterns—are deterministically bounded or known a-priori. CMP real-time schedulers that target this space provide *schedulability utilization bounds*, $U$, below which all tasks meet their deadlines. Examples include the Pfair and LLREF algorithms with $U = m$ and the global EDF algorithms with $U \approx m/2$ on an $m$-processor CMP [3]. Although this is an extremely important real-time application space, there also exist some real-time applications with behaviors outside this envelope — e.g., those with uncertainties on task arrival, execution-time, and resource access behaviors, caused due to data- and context-dependent executions, resulting in overloads (i.e., $U > m$) [5]. During overloads, such applications typically desire "best-effort" timing assurance in the sense that processor cycles are assured to be allocated in a way that optimizes, and enables the reasoning of, collective application timeliness — as specified by the application [9]. Best-effort real-time scheduling requires, among other things, a two-way interaction between the application and OS kernel, especially for asynchronously and safely aborting selected tasks.

Our primary motivation in this paper is to create an OS platform that can support best-effort real-time schedulers on CMPs. When creating an RTOS platform, the open-source Linux kernel is often a compelling base platform choice. Linux is one of the most advanced operating systems with wide acceptance in industry, used across various form factors — from enterprise level servers to embedded devices.

In the past, efforts have been made to provide real-time extensions to the Linux kernel. Two popular extensions, RTAI and Xenomai, are built around the idea of Linux running as a task above the ADEOS nanokernel [1]. Both have the downside that real-time tasks are written to the APIs of the extension, rather than standard Linux APIs. More recently, the Linux kernel team's `PREEMPT_RT` real-time patch enables complete kernel preemption and improves interrupt latencies, thus allowing real-time tasks to execute within Linux itself. The Linux kernel augmented with the `PREEMPT_RT` patch is therefore a compelling RTOS platform.

Thus, our goal is to create an OS platform with the following properties:

**P1**: The platform must support the implementation of

best-effort real-time schedulers.

**P2**: The platform must support CMP real-time scheduling.

**P3**: The platform must be based on the Linux kernel.

**P4**: The platform must use the `PREEMPT_RT` patch.

We present ChronOS Linux, which provides these properties. ChronOS Linux (hereafter refered to as ChronOS) supports the notion of real-time scheduling segments that facilitate the expression and enforcement of time constraints, and asynchronous and safe task termination. ChronOS provides a scheduling framework for the implementation of a broad range of CMP real-time schedulers, best-effort, non-best-effort, global, and partitioned, as scheduler plugins. We implemented a suite of schedulers in ChronOS including G-EDF, G-NP-EDF, P-EDF [3], P-DASA [6], and GUA [7] on a quad-core AMD Phenom platform. Our experiments reveal that the scheduler behaviors are consistent with their theoretical behaviors, and the effectiveness of ChronOS.

Past efforts that overlap with a subset of ChronOS's problem space include the Alpha OS [9], RED-Linux [11], and LITMUS$^{RT}$ [3]. Alpha, which pioneered the best-effort real-time concept, is not Linux-based. RED-Linux supports fixed and dynamic priority scheduling, but does not support CMPs. LITMUS$^{RT}$ provides extensive support for CMP scheduling on Linux, but it does not support best-effort real-time scheduling. Neither RED-Linux nor LITMUS$^{RT}$ use the `PREEMPT_RT` patch. Thus, to the best of our knowledge, ChronOS is the only Linux/`PREEMPT_RT` kernel that supports CMP real-time scheduling including best-effort scheduling — the paper's contribution.

# 2. PREEMPT_RT REAL-TIME PATCH

The stock Linux kernel provides soft real-time capabilities, such as the POSIX primitives and the ability to set priorities. The kernel provides two real-time scheduling policies — `SCHED_FIFO` and `SCHED_RR` and a "nice" based scheduling policy called `SCHED_NORMAL`. `SCHED_FIFO` executes processes in FIFO order within a priority and will only preempt a task to execute a newly arrived task at a higher priority. This is primarily a POSIX-specified feature. `SCHED_RR`, on the other hand, schedules processes of the same priority in a round robin fashion while favoring higher priority tasks.

In order to bring in real-time semantics to the kernel, it is required to have a preemptible kernel. To achieve this we use the `PREEMPT_RT` patch which enables complete kernel preemption along with a generic clock event layer with high resolution support. The patch improves interrupt latencies and allows preemption of most parts of the Linux kernel except for a few regions which are inherently unpreemptible, such as the task scheduler. In-kernel locking primitives have been re-implemented using rtmutexes. As a result, critical sections that are protected with `spinlock_t` or `rwlock_t` are preemptible. Priority inheritance has been implemented for in-kernel spinlocks and semaphores. Interrupt handlers in the kernel have been converted into preemptible kernel threads. The Linux timer APIs have been converted into separate infrastructure for high resolution kernel timers.

# 3. CHRONOS REAL-TIME SCHEDULER

## 3.1 Priority Bit-map and Run-queues

There are 140 priority levels in Linux, represented as a per-processor bit-map. $[0 \ldots 99]$ are referred to as real-time priorities while $[100 \ldots 139]$ are called "nice" priorities. In the kernel-space, "0" is the highest real-time priority while "99" is the least (which is opposite to that in the user-space). Each CPU has a run-queue of active tasks at a priority. The scheduler starts from the highest priority bit in the processor's bit-map, looks for tasks at that priority level and executes them before going to the next level.

The use of the `PREEMPT_RT` patch significantly complicated scheduling, because interrupts now reside in the same priority space as real-time tasks. We therefore cannot take the same approach as Litmus and place our tasks in a priority band higher than all other tasks in the system, because this would block interrupts, which we may need. In order to facilitate working on the real-time tasks, for every priority level in the bit-map we create another queue called the ChronOS real-time run-queue (`CRT-RQ`) which holds references to the ChronOS real-time tasks in the Linux run-queue. The `CRT-RQ` is therefore always a subset of the Linux run-queue. Tasks can therefore be given a priority representing their dependence on various interrupts.

## 3.2 Scheduling Real-Time Tasks

The ChronOS scheduler extends the Linux scheduler. Every ChronOS real-time task starts as a Linux real-time task scheduled under `SCHED_FIFO`. It becomes a ChronOS real-time task by entering a real-time segment. A real-time segment is defined as a portion of the thread which needs to be executed with real-time time constraints. This can be done using the following system calls in ChronOS: `begin_rt_seg()` is used to indicate the start of a real-time scheduling segment and also to provide the real-time timing constraints for a given task. `end_rt_seg()` indicates the end of a real-time scheduling segment. We add additional parameters to the `struct task_struct` to specify the real-time properties of a task, such as the task's worst case execution cost `(WCET)`, deadline, period and time-utility function `(TUF)`. These are set during the `begin_rt_seg()` system call.

To schedule, we hook into the existing Linux `SCHED_FIFO` scheduler. After `SCHED_FIFO` has found the highest priority, we call the ChronOS scheduler. When a ChronOS scheduler is enabled and the `CRT-RQ` for that priority is not empty, the ChronOS scheduler selects a task from the `CRT-RQ` based on the scheduling algorithm selected, and returns the task to `SCHED_FIFO` for execution. Therefore, a ChronOS real-time task executes before a Linux real-time task of the same priority, but after a Linux real-time task of a greater priority.

The real-time scheduler is invoked at various scheduling events. A scheduling event is defined as a trigger that forces the system into a scheduling cycle resulting in a call to the scheduler, which selects a new task. In ChronOS we define the following scheduling events.

**Task begins a segment** – When a task begins its segment, it is added to the `CRT-RQ` and the scheduler is invoked. At that time, the scheduling algorithm selects the best task from the `CRT-RQ` and returns it to `SCHED_FIFO`.

**Task ends a segment** – When a task finishes its scheduling segment, the task is removed from the `CRT-RQ` and the scheduler is invoked.

**Resource is requested** – When a task requests for a resource, ChronOS tags the task as `RESOURCE_REQUESTED` and invokes the scheduler. This is done to let the scheduling algorithm look at the dependency chain based on the resource

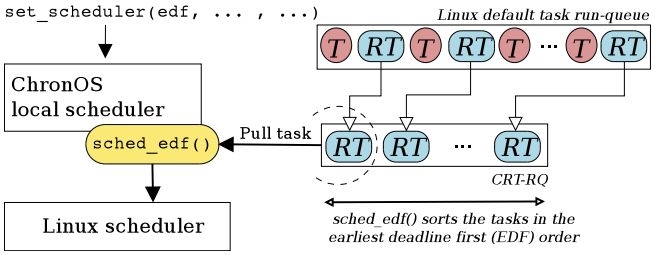


**Figure 1: ChronOS scheduling approach on a single-processor system**

requested and pick the task that is best suited for execution.

**Resource is released** – When a task releases a resource, ChronOS invokes the scheduler in order to allow a new task to be picked which might be blocking on the resource that was just released. The decision to choose the new task is done by the scheduling algorithm.

A scheduling algorithm is selected via the `set_scheduler()` system call. All scheduling algorithms are created as Linux modules in ChronOS which provides the flexibility to add or remove any scheduling algorithm from a running kernel without restarting the system. The scheduling algorithms are implemented in a modular fashion using a set of functions that we refer to as the "scheduler plugin". Once a scheduler is selected for a set of processors, all the real-time tasks that are added to the system on those processors are scheduled using the the selected scheduling algorithm.

# 4. SINGLE PROCESSOR SCHEDULING

Fig. 1 illustrates an example of scheduling on a single processor machine. As the scheduling algorithms are written as modules in ChronOS, they can be dynamically loaded into a running kernel. This adds the scheduling algorithms to the list of available schedulers. In Fig. 1, the call to `set_scheduler()` is made from the real-time application to select `EDF` as the real-time scheduler. ChronOS checks if the `EDF` kernel module is available. If the scheduler is found, ChronOS loads the plugin and makes it the default ChronOS local scheduler for running real-time tasks. All the real-time tasks are now added to the `CRT-RQ` at their deadline position. At every scheduling event, the ChronOS scheduler invokes `sched_edf()`. Since items are inserted in order, the head of the `CRT-RQ` now represents the earliest deadline task, which is returned to `SCHED_FIFO` for execution. For algorithms that use dynamic keys, such as `TUF`-based algorithms, the scheduler would have to sort the `CRT-RQ`.

# 5. MULTIPROCESSOR SCHEDULING

Scheduling on multiprocessors can be mainly categorized into three forms – partitioned, clustered, and global scheduling. ChronOS supports all these variants. Since clustering is simply a combination of the two, we only describe the details of partitioned and global architectures.

## 5.1 Partitioned Scheduling

Partitioned scheduling can be described as uniprocessor scheduling done on multiprocessors. The key idea of partitioned scheduling is to divide the task-set using an off-line heuristic (such as first-fit or best-fit) to partitioning a set of tasks on $M$ processors. This has been shown to be equivalent to the bin-packing problem [10] and hence NP-hard in the strong sense. Baruah *et al.* present a polynomial-time algorithm to partition collection of sporadic tasks onto $M$ processors in [2].

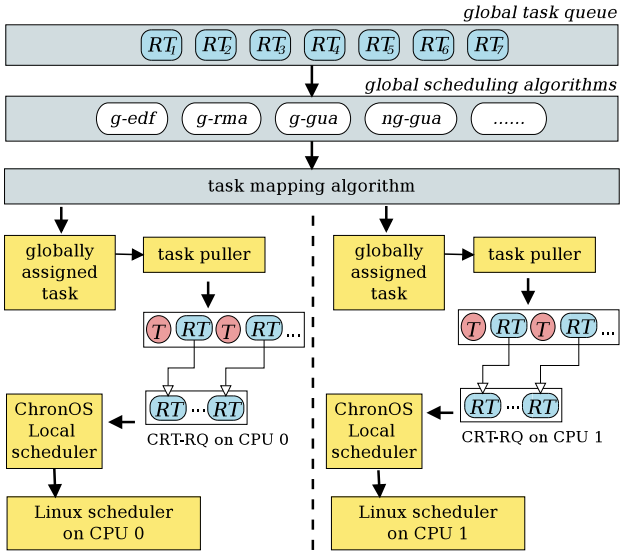


**Figure 2: ChronOS global scheduling approach on a multiprocessor system**

To use partitioned scheduling algorithms on ChronOS, we partition the task-set using off-line polynomial-time heuristics. For an $M$ processor system, the heuristics divides the task-sets into $M$ processor bins. Once all the tasks have been divided, the affinity of each of the tasks is set to the processor they have been assigned to. The tasks are added to the `CRT-RQ` of their respective assigned processors. As partitioned scheduling is an extension of uniprocessor scheduling, the partitioned scheduler is set as the local ChronOS scheduler on all processors using the `set_scheduler()` system call. Each processor runs its scheduling algorithm independently. At every scheduling event, the processor enters its local scheduler, looks at the local `CRT-RQ`, and picks the next task to be executed.

## 5.2 Global Scheduling

Most of the multiprocessor scheduling algorithms, such as `G-EDF`, `Pfair` and `GUA` are based on global scheduling. The main idea behind global scheduling is that the tasks are assigned to a global queue instead of individual local queues. The scheduling algorithm on each processor looks at the global queue to make a scheduling decision. Clustering is a simple extension of global scheduling wherein multiple global scheduling domains are created, each of which spans some subset of the systems processors.

Fig. 2 illustrates the global scheduling approach used in ChronOS. In order to implement global scheduling, we create another level of scheduling abstraction. At the top we have the "global scheduler" which looks at the "global task queue". The global scheduler maps to a "local scheduler" on individual processors which extends from `SCHED_FIFO`. The global scheduler (invoked on a processor) either picks the top task or the top $M$ tasks, depending on its type. In the latter case, these tasks are given to the task mapping algorithm which maps these tasks on $M$ underlying processors. The tasks assigned by the task mapping algorithm are pushed into the "globally assigned task" block from where the "task puller" on each CPU picks up the task and moves it to the

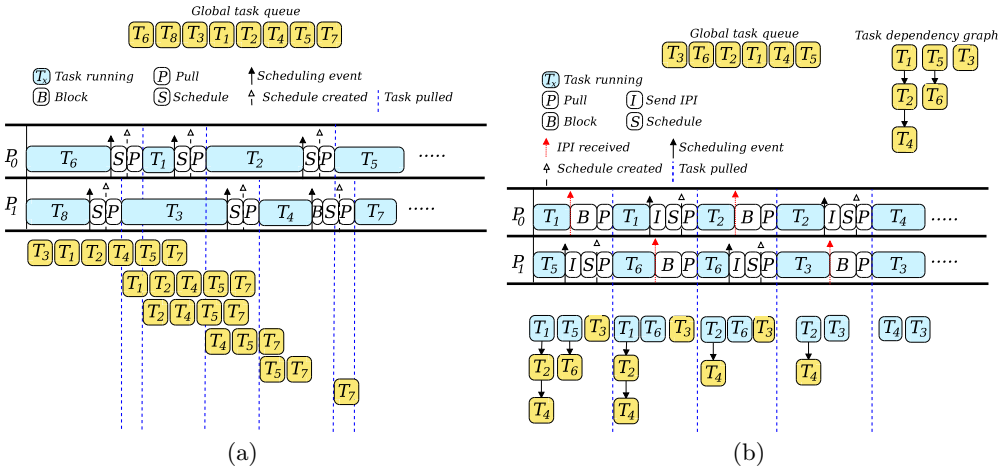


**Figure 3: Global scheduling architecture models in ChronOS (a) "Application concurrent" model (b) "Stop-the-World" model**

head of its local queue. The default local scheduling algorithm for global scheduling algorithms on each processor is FIFO, which picks the head of the `CRT-RQ` queue and gives the task to `SCHED_FIFO` for execution.

There are two ways in which global scheduling can be achieved. In the "Application Concurrent" scheduling model, the global scheduler (such as `G-NP-EDF`) picks a task for itself. In the "Stop-the-World"(STW) scheduling model, the global scheduler (such as Pfair or GUA) picks the tasks for all $M$ available processors.

### 5.2.1 *Application Concurrent Scheduling Model*

Fig. 3(a) illustrates the application concurrent scheduling model. For the sake of explanation of the model, we will assume that the scheduling algorithm picks the task at the head of the queue, and task execution is non-preemptible. At the beginning, task $T_6$ is running on processor $P_0$ and task $T_8$ is running on processor $P_1$. As shown, $T_8$ finishes before $T_6$. However, $T_6$ is not preempted on $P_0$. After $T_8$ finishes, it generates a scheduling event. $P_1$ enters the global scheduler, picks the first eligible task $T_3$ from the global queue and assigns it to $P_1$. The local scheduler on $P_1$ pulls the task $T_3$ and starts executing it. While $P_1$ pulls its task, $T_6$ finishes on $P_0$ and generates a scheduling event. It pulls $T_1$ from the global queue and starts executing it without preempting $T_3$ on $P_1$. The same procedure is repeated for other scheduling events.

There might be a scenario when both processors finish their tasks concurrently. As shown in the Fig. 3(a), when $T_2$ finishes on $P_0$, $T_4$ finishes on $P_1$. As the global task queue is shared between processors, while $P_0$ enters the global scheduler, $P_1$ blocks. When $P_0$ finishes, $P_1$ unblocks and enters the scheduler. There may also be a case where $P_1$ is not executing a real-time task when $P_0$ schedules. In such a case, an "Inter-processor Interrupt" (`IPI`) is sent to $P_1$ to force it to reschedule.

G-NP-EDF and G-FIFO are some of the algorithms that use such an architecture model. The downside of the application concurrent model is that it is difficult to implement scheduling algorithms that consider resource dependencies. Additionally, since the schedule is only generated for a single processor, there is no way to implement optimal algorithms such as LLREF or PFair.

### 5.2.2 *Stop-the-World Scheduling Model*

In order to allow optimal global scheduling algorithms and resource management, ChronOS implements the STW scheduling model. Fig. 3(b) provides an illustration of the model. The figure shows the dependency relation of the tasks in the global task queue with each other. Task $T_4$ needs a resource owned by task $T_2$ which in turn requires a resource held by task $T_1$. In a similar fashion, task $T_6$ needs a resource owned by task $T_5$. Task $T_3$ does not have any dependents. Let us assume that the scheduling algorithm considers the tasks that have the maximum dependents as the most eligible tasks for execution.

Fig. 3(b) shows that tasks $T_1$ and $T_5$ are currently executing on processors $P_0$ and $P_1$, respectively. Task $T_5$ finishes first and generates a scheduling event. In the STW model, once a scheduling event is generated, the schedule needs to be created for all the processors. This requires a processor to be able to force a scheduling event on all other processors. When task $T_5$ triggers the scheduling event, processor $P_1$ sends an (`IPI`) to all the processors which forces the processors to schedule.

In the example shown in Fig. 3(b), processor $P_1$ sends an `IPI` to all processors. After sending the `IPI`, $P_1$ enters its global scheduler and looks at the available tasks in the run-queue to create the schedule. Meanwhile, processor $P_0$ receives the `IPI` and is forced into the scheduler. However, as $P_1$ is already in the global scheduler, $P_0$ blocks. The global scheduler on $P_1$ picks two eligible tasks (assuming a two-processor system in the example) and hands these tasks to the mapping algorithm. The task mapper pushes these tasks into the "globally assigned task" block of each processor. Each processor pulls its assigned task to the head of its local queue which is then executed by the Linux scheduler.

## 5.3 Mapping Tasks to Processors

Fig. 4 illustrates the mapping algorithm used in ChronOS for global scheduling. The job of the algorithm is to take the $M$ most eligible tasks that have been selected by the globlal scheduler and map them to the $M$ available processors such that task migrations are reduced and cache coherence is preserved. The algorithm shown in Fig. 4 is selected as the default for global scheduling algorithms. However, the default task-mapper can be overridden in ChronOS.

The mapping is done using a three-pass algorithm. In Fig. 4 tasks $RT_5$, $RT_8$, $RT_1$ and $RT_4$ are four real-time tasks that have been selected by the global scheduling algorithm. Each run-queue shows the tasks that belong to the individual processors. We also highlight the current running task on each processor before the scheduling event was triggered. Task $RT_2$ is the current running task on processor $P_0$, task $RT_5$ on $P_1$, task $RT_9$ on $P_2$, and task $RT_7$ on $P_3$.

In the first pass, the algorithm examines the final schedule and maps currently running tasks to the processors they are running on. As shown in Fig. 4, task $RT_5$, the current running task on $P_1$, is mapped to $P_1$. In the second pass, the algorithm examines the final schedule and maps tasks to the processor they are currently on. This prevents tasks from being unnecessarily migrated. As shown, since task $RT_4$ resides on processor $P_0$, it is mapped there. In the similar fashion, task $RT_8$ is mapped to $P_2$. In the last pass, the leftover tasks are mapped to the remaining processor(s). As shown, task $RT_1$ is assigned to processor $P_3$. Since task $RT_1$ resides on processor $P_0$, this step results in the migration of the mapped tasks.

If the final schedule consists of $M$ tasks that all belong to the same processor, the worst case migration cost is $M - 1$ migrations. In such cases, cache-aware scheduling algorithms can be used to create a cache conscious schedule which limits migrations. Guan *et al.* [8] and Calandrino *et al.* [4] present cache-aware real-time scheduling algorithms.

# 6. EXPERIMENTAL EVALUATION

To evaluate the performance of ChronOS, we conducted experiments on a quad-core AMD Phenom machine. We measured overheads associated with real-time scheduling and experimentally compared the scheduling algorithms previously listed with their theoretical results.

## 6.1 NG-GUA and G-GUA

NG-GUA and G-GUA are multiprocessor polynomial-time heuristic scheduling algorithms that allow tasks to be subject to run-time uncertainties, overloads and dependencies, and yield optimal total utility in underload and best-effort timeliness behavior based on TUF scheduling otherwise. NG-GUA defaults to G-EDF during under-loads while G-GUA does not. G-EDF and NG-GUA should therefore meet all deadlines until `m/2`, while G-GUA may miss deadlines earlier. Once G-GUA and NG-GUA begin missing deadlines, both should be free from the "domino effect" commonly seen

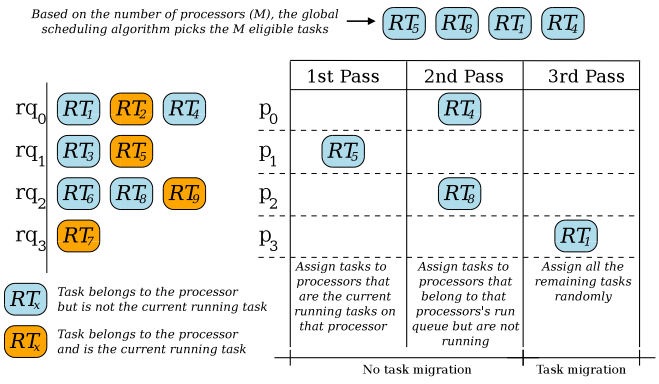


**Figure 4: Task mapping for global scheduling algorithms in ChronOS**

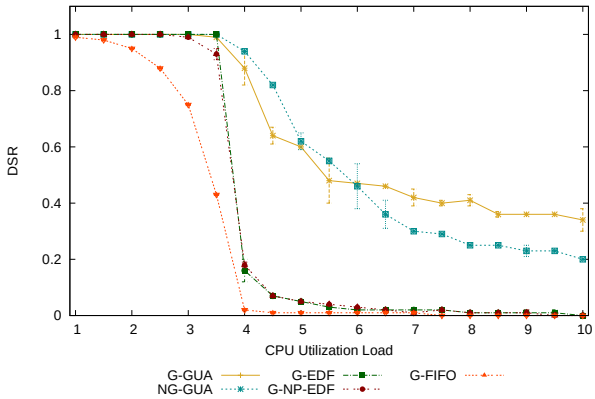


**Figure 5: Deadline satisfaction ratio (DSR)**

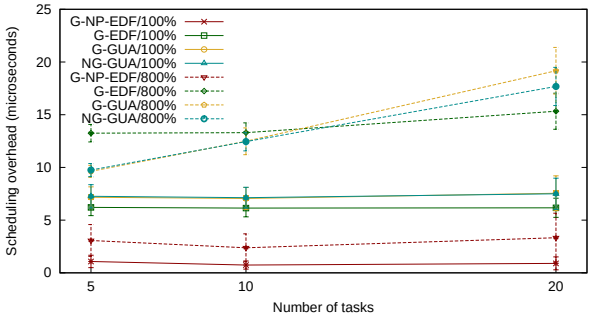


**Figure 6: ChronOS scheduling overheads**

in EDF-type algorithms. For brevity, the details of the algorithms have been omitted. They are given by Garyali in [7].

## 6.2 Scheduling Results

We create a real-time test application using ChronOS APIs which periodically fires real-time tasks with specified time-constraints. For each task, we use a `burn_cpu(wcet)` function, which takes the `wcet` as an input and burns processor cycles for that amount of time. To simplify experimentation, we use periodic tasks with deadlines equal to periods. Fig. 5 shows deadline satisfaction ratio (DSR) results for several best-effort and non-best-effort scheduling algorithms. The results shown are for a 12-task taskset with periods distributed between 40ms and 2s, and randomly generated utilities and execution times. We observe that the results are consistent with the theoretical behavior of the algorithms; G-EDF and G-NP-EDF demonstrate the "domino effect" when overloaded, while the TUF based algorithms do not. G-EDF and NG-GUA also exhibit identical performance in underloads, as expected. More extensive scheduling results are presented by Garyali in [7].

## 6.3 Overheads

To measure the overhead incurred by our scheduling framework over `SCHED_FIFO`, we measured four sources of overhead: context switching overhead, scheduling overhead, in-scheduler migration overhead, and the overhead of our real-time specific system calls. All times measured, both in the kernel and in the userspace, were taken by reading the x86 architecture's Time Stamp Counter (TSC) before and after the event measured. All results are from at least 1000 runs.

Context switching overhead describes the cost of the Linux kernel's `context_switch()` function, which performs the actual switching of the old and new tasks. The function is also

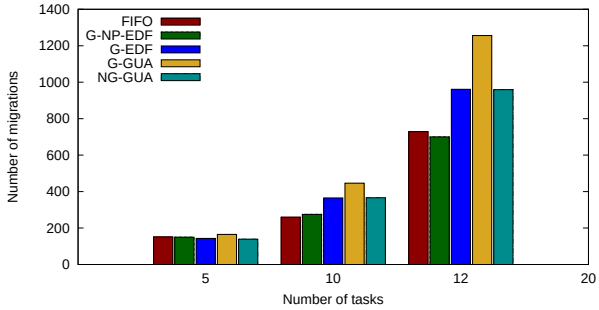


**Figure 7: Number of migrations**

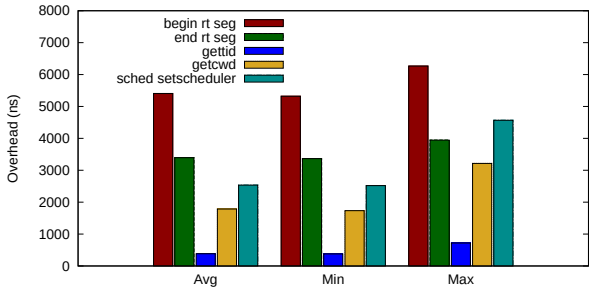


**Figure 8: ChronOS system call overheads**

called by the Linux scheduler and is therefore not specific to ChronOS, but due to the fully preemptive nature of certain scheduling algorithms (such as NG-GUA, G-GUA), context switches are much more frequent under these algorithms than under `SCHED_FIFO`. Also, a context switch under such algorithms is more expensive, since the task that is being switched in often has been preempted from another processor, incurring cache misses. The average context switching overhead under `SCHED_FIFO` was 3.7$\mu$s with a standard deviation of 1.4$\mu$s. Under G-GUA, the average time was 12.7$\mu$s with a standard deviation of 2.0$\mu$s.

Scheduling overheads represent the time required to actually run the scheduling code for a specific algorithm. Scheduling overheads are shown for various algorithms, numbers of tasks, and utilizations in Fig. 6. Scheduling overheads are expectedly high at low load for the more complex GUA family of algorithms, due to their more complex task selection scheme. Likewise, at high loads non-abortive algorithms such as G-EDF display a higher overhead, since they backlog tasks, rather than aborting and removing them.

In-scheduler migration reflects the time required by the kernel scheduler to pull a specific task from a remote run-queue to the local run-queue. This is largely dominated by the time required to lock the necessary spinlocks. We found that that average in-kernel task migration overhead in ChronOS is 8$\mu$s, with a standard deviation of 3$\mu$s. We also recorded the number of such migrations performed by each algorithm on various task-sets. These are displayed in Fig. 7. As expected, G-EDF and NG-GUA (which defaults to G-EDF) performs a significantly higher number of migrations than G-NP-EDF, which is similar to `SCHED_FIFO`. G-GUA performs by far the most migrations on a given task-set. Therefore, while on a per-migration basis the cost is constant, G-GUA suffers much higher migration overhead.

We recorded the overheads for two system calls used by our real-time tasks to begin and end real-time scheduling segments. In Fig. 8 they are compared to the time required by three standard Linux system calls, `gettid()`, `getcwd()`, and `sched_setscheduler()`. Since `gettid()` simply returns a value from the task descriptor of the current task, it is a reasonable estimate of system call overhead. As shown, our system calls are comparatively long. The majority of this time is required to perform a call to the Linux scheduler.

# 7. CONCLUSIONS

We have presented ChronOS, a Linux/`PREEMPT_RT` OS that supports CMP real-time scheduling including best-effort scheduling. At its core, ChronOS demonstrates that the best-effort timeliness notion, pioneered by the Alpha kernel, can be supported on Linux/`PREEMPT_RT` CMP platforms.

Our future work will focus on understanding and increasing the scalability of global schedulers in ChronOS on high core-count systems. This paper is based on ChronOS Beta 2.4 and Linux kernel 2.6.31.14. ChronOS is open-sourced under GPL and is publicly available through a website, withheld here to respect the conference anonymity rules.



# 8. REFERENCES

[1] A. Barbalace, A. Luchetta, G. Manduchi, M. Moro, A. Soppelsa, and C. Taliercio. Performance Comparison of VxWorks, Linux, RTAI, and Xenomai in a Hard Real-Time Application. *Nuclear Science, IEEE Transactions on*, 55(1):435 –439, feb. 2008.

[2] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *RTSS '05: Proceedings of the 26th IEEE International Real-Time Systems Symposium*, pages 321–329, 2005.

[3] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leon-tyev, and J. Anderson. LITMUS$^{RT}$: A Status Report. In *RTLWS '07*, 2007.

[4] J. M. Calandrino and J. H. Anderson. On the design and implementation of a cache-aware multicore real-time scheduler. In *ECRTS '09*, pages 194–204, 2009.

[5] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley. An Adaptive, Distributed Airborne Tracking System ("Process the Right Tracks at the Right Time"). In *In IEEE WPDRTS, volume 1586 of LNCS*, pages 353–362. Springer-Verlag, 1999.

[6] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, CMU, 1990. CMU-CS-90-155.

[7] P. Garyali. On best-effort utility accrual real-time scheduling on multiprocessors. Master's thesis, Virginia Tech, 2010. http://scholar.lib.vt.edu/theses/available/etd-07222010-114202/.

[8] N. Guan, M. Stigge, W. Yi, and G. Yu. Cache-aware scheduling and analysis for multicores. In *EMSOFT '09*, pages 245–254, 2009.

[9] E. Jensen and J. Northcutt. Alpha: A Non-proprietary OS for Large, Complex, Distributed Real-Time Systems. In *Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on*, pages 35–41, 11-12 1990.

[10] D. S. Johnson. Fast algorithms for bin packing. *J. Comput. Syst. Sci.*, 8(3):272–314, 1974.

[11] Y.-C. Wang and K.-J. Lin. Implementing a general real-time scheduling framework in the red-linux real-time kernel. In *RTSS '99*, page 246, 1999.